UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD-A208 515

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

2

1. REPORT NUMBER

12. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

Ada Compiler Validation Summary Report: Encore Computer Corporation, Encore Verdix Ada Development System Version 5.5, Encore Multimax 320 (Host & Target) 890113S1.09161

5. TYPE OF REPORT & PERIOD COVERED

09 Jan. 1989- 30 Nov. 1989

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

National Institute of Standards and Technology
Gaithersburg, Maryland, USA

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION AND ADDRESS

National Institute of Standards and Technology
Gaithersburg, Maryland, USA

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS

Ada Joint Program Office
United States Department of Defense
Washington, DC 20301-3081

12. REPORT DATE

09 Jan 1989

13. NUMBER OF PAGES

14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office)

National Institute of Standards and Technology
Gaithersburg, Maryland, USA

15. SECURITY CLASS (of this report)

UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20. If different from Report)

UNCLASSIFIED

DTIC
ELECTE
MAY 25 1989
S D
D

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Encore Verdix Ada Development System, Version 5.5, Encore Computer Corporation, National Institute of Standards and Technology, Encore Multimax 320 under Umax V, Version R2.2, (Host and Target), ACVC 1.09

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73 S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AVF Control Number: NIST89ENC515_2_1.09
01-13-89:02-09-89

Ada Compiler
VALIDATION SUMMARY REPORT:
Certificate Number: 890113S1.09161
Encore Computer Corporation
Encore Verdix Ada Development System, Version 5.5
Encore Multimax 320 Host; Encore Multimax 320 Target

Completion of On-Site Testing:
January 09, 1989

Prepared By:
Software Standards Validation Group
National Computer Systems Laboratory
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, Maryland 20899

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

Ada Compiler Validation Summary Report:

Compiler Name: Encore Verdix Ada Development System, Version 5.5

Certificate Number: 890113S1.09161

Host:
Encore Multimax 320 under
Umax V,
Version R2.2

Target:
Encore Multimax 320 under
Umax V,
Version R2.2

Testing Completed January 09, 1989 Using ACVC 1.9

This report has been reviewed and is approved.

Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division
National Computer Systems Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria, VA 22311

Ada Joint Program Office
Mr. William S. Ritchie, Acting Director
Department of Defense
Washington DC 20301

Ada Compiler Validation Summary Report:

Compiler Name: Encore Verdix Ada Development System, Version 5.5

Certificate Number: 890113S1.09161

Host:
Encore Multimax 320 under
Umax V,
Version R2.2

Target:
Encore Multimax 320 under
Umax V,
Version R2.2

Testing Completed January 09, 1989 Using ACVC 1.9

This report has been reviewed and is approved.

Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division
National Computer Systems Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria, VA 22311

Ada Joint Program Office
Mr. William S. Ritchie, Acting Director
Department of Defense
Washington DC 20301

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

This information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of test are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

> To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
>
> To attempt to identify any unsupported language constructs required by the Ada Standard
>
> To determine that the implementation-dependent behavior is allowed by the Ada Standard

On-site testing was completed January 09, 1989 at Marlborough, MA .

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

> Ada Information Clearinghouse
> Ada Joint Program Office
> OUSDRE
> The Pentagon, Rm 3D-139 (Fern Street)
> Washington DC 20301-3081

or from:

> Software Standards Validation Group
> National Computer Systems Laboratory
> National Institute of Standards and Technology
> Building 225, Room A266
> Gaithersburg, Maryland 20899

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines. Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide., December 1986.

1.4 DEFINITION OF TERMS

ACVC — The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.

Ada Commentary — An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.

Ada Standard — ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

Applicant — The agency requesting validation.

AVF — The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.

AVO — The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.

Compiler — A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and

interpreters.

Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Language Maintenance	The Language Maintenance Panel (LMP) is a committee established by the Ada Board to recommend interpretations and Panel possible changes to the ANSI/MIL-STD for Ada.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	An Ada program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler.

A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then

the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: Encore Verdix Ada Development System, Version 5.5

ACVC Version: 1.9

Certificate Number: 890113S1.09161

Host Computer:

Machine:	Encore Multimax 320
Operating System:	Umax V, Version R2.2
Memory Size	28 MBytes

Target Computer:

Machine:	Encore Multimax 320
Operating System:	Umax V, Version R2.2
Memory Size:	28 MBytes

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

- Capacities.

  The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See test D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

- Universal integer calculations.

  An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX_INT. This implementation processes 64 bit integer calculations. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

- Predefined types.

  This implementation supports the additional predefined types SHORT_INTEGER, SHORT_FLOAT, and TINY_INTEGER in the package STANDARD. (See tests B86001BC and B86001D.)

- Based literals.

  An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR or CONSTRAINT_ERROR during execution. This implementation raises NUMERIC_ERROR during execution. (See test E24101A.)

- Expression evaluation.

  Apparently no default initialization expressions for record components are evaluated before any value is checked to belong

to a component's subtype. (See test C32117A.)

Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

This implementation uses no extra bits for extra precision. This implementation uses all extra bits for extra range. (See test C35903A.)

Apparently NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

Sometimes NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

Apparently underflow is not gradual. (See tests C45524A..Z.)

- Rounding.

The method used for rounding to integer is apparently round to even. (See tests C46012A..Z.)

The method used for rounding to longest integer is apparently round to even. (See tests C46012A..Z.)

The method used for rounding to integer in static universal real expressions is apparently round to even. (See test C4A014A.)

- Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises no exception. (See test C36003A.)

NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. (See test C36202A.)

NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with SYSTEM.MAX_INT + 2 components. (See test C36202B.)

A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR when the array type is declared. (See test

C52103X)

A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array type is declared. (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications. (See test E38104A.)

In assigning record types with discriminants, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Aggregates.

In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

All choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

- Representation clauses.

An implementation might legitimately place restrictions on representation clauses used by some of the tests. If a representation clause is not supported, then the implementation must reject it.

Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are supported. (See tests C35502I..J, C35502M..N, and A39005F.)

Enumeration representation clauses containing noncontiguous values for character types are supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

Enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1) are supported. (See tests C35508I..J and C35508M..N.)

Length clauses with SIZE specifications for enumeration types are supported. (See test A39005B.)

Length clauses with STORAGE_SIZE specifications for access types are supported. (See tests A39005C and C87B62B.)

Length clauses with STORAGE_SIZE specifications for task types are supported. (See tests A39005D and C87B62D.)

Length clauses with SMALL specifications are supported. (See tests A39005E and C87B62C.)

Record representation clauses are not supported. (See test A39005G.)

Length clauses with SIZE specifications for derived integer types are supported. (See test C87B62A.)

- Pragmas.

The pragma INLINE is supported for procedures. The pragma INLINE is supported for functions. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

- Input/output.

The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D and CE2102E.)

Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I, and CE2102J.)

RESET and DELETE are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2102G and CE2102K.)

Dynamic creation and deletion of files are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2106A and CE2106B.)

Overwriting to a sequential file truncates the file to last element written. (See test CE2208B.)

An existing text file can be opened in OUT_FILE mode, can be created in OUT_FILE mode, and can be created in IN_FILE mode. (See test EE3102C.)

More than one internal file can be associated with each external file for text I/O for both reading and writing. (See tests CE3111A..E (5 tests), CE2110B, and CE2111D.)

More than one internal file can be associated with each external file for sequential I/O for both reading and writing. (See tests CE2107A..D (4 tests), CE2110B, and CE2111D.)

More than one internal file can be associated with each external file for direct I/O for both reading and writing. (See tests CE2107F..I (4 tests), CE2110B, and CE2111H.)

An external sequential access file and an internal direct access file can be associataed with a single external file for writing. (See test CE2107E.)

An external file associated with more than one internal file can be deleted for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO. (See test CE2110B.)

Temporary sequential files are given names. Temporary direct files are given names. Temporary files given names are deleted when they are closed. (See tests CE2108A and CE2108C.)

- Generics.

Generic subprogram declarations and bodies can compiled in separate compilations. (See tests CA1012A and CA2009F.)

Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.9 of the ACVC comprises 3122 tests. When this compiler was tested, 28 tests had been withdrawn because of test errors. The AVF determined that 226 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for withdrawn tests. Modifications to the code, processing, or grading for 26 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	109	1049	1630	17	17	46	2868
Inapplicable	1	2	223	0	0	0	226
Withdrawn	3	2	21	0	2	0	28
TOTAL	113	1053	1874	17	19	46	3122

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	190	499	540	245	165	98	142	326	137	36	234	3	253	2868
Inapplicable	14	73	134	3	0	0	1	1	0	0	0	0	0	226
Withdrawn	2	14	3	0	1	1	2	0	0	0	2	1	2	28
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122

3.4 WITHDRAWN TESTS

The following 28 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

B28003A	E28005C	C34004A	C35502P	A35902C	C35904A
C35904B	C35A03E	C35A03R	C37213H	C37213J	C37215C
C37215E	C37215G	C37215H	C38102C	C41402A	C45332A
C45614C	E66001D	A74106C	C85018B	C87B04B	CC1311B
BC3105A	AD1A01A	CE2401H	CE3208A		

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 226 test were inapplicable for the reasons indicated:

C35702B uses LONG_FLOAT which is not supported by this implementation.

A39005G uses length clauses with SIZE specifications for enumeration types which are not supported by this compiler.

The following (13) tests use LONG_INTEGER, which is not supported by this compiler.

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45631C	C45632C
B52004D	C55B07A	B55B09C		

C45531M, C45531N, C45532M, and C45532N use fine 48-bit fixed-point base types which are not supported by this compiler.

C45531O, C45531P, C45532O, and C45532P use coarse 48-bit fixed-point base types which are not supported by this compiler.

C86001F redefines package SYSTEM, but TEXT_IO is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependent on the package TEXT_IO.

C96005B requires the range of type DURATION to be different from those of its base type; in this implementation they are the same.

The following 201 tests require a floating-point accuracy that exceeds the maximum of 15 digits supported by this implementation:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into sub-tests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 26 Class B tests.

The following Class B tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B24009A	B24204A	B24204B	B24204C	B2A003A
B2A003B	B2A003C	B33301A	B37201A	B38003A

B38003B	B38009A	B38009B	B41202A	B44001A
B64001A	B67001A	B67001B	B67001C	B67001D
B91001H	B91003B	B95001A	B97102A	BC1303F
BC3005B				

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 *Prevalidation*

Prior to validation, a set of test results for ACVC Version 1.9 produced by the Encore Verdix Ada Development System was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the Encore Verdix Ada Development System using ACVC Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of a Encore Multimax 320 operating under Umax V, Version R2.2.

A magnetic tape containing all tests except for withdrawn tests was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. The magnetic tape was read with a UNIX utility, ansitape, which read the ANSI standard tape into the UNIX system. Some of the files on the tape contained multiple ACVC test sources and a NIST supplied utility, UNPACK.ADA, was used to "unpack" those files such that there was one ACVC source program per file. The files were then moved using a UNIX command, rcp, to their respective directories on the disk of the HOST/TARGET hardware.

In a few .tst files, the macro substitutions that were done by NIST were incompatible with the limitations of the UMAX operating system. The macro substitutions were done again during the validation with values compatible with UMAX. The affected files are:

C35502D	C35502F	CE2102B	CE2102C	CE2102H	CE2103A
CE2103B	CE2107A				

Split tests as supplied by Encore were checked against those corresponding tests from the magnetic tape using a UNIX utility, diff. No differences were found except those expected. The split tests as supplied by Encore were used in the validation.

After the test files were loaded to disk, the full set of tests was compiled and linked on the Encore Multimax 320, and all executable tests were linked and run. Results were printed from the target computer via

UNIX remote line printer facility over Ethernet.

The compiler was tested using command scripts provided by Encore Computer Corporation and reviewed by the validation team. The compiler was tested using all default option settings except for the following:

Option | Switch	Effect
"-M"	creates an executable program called "a.out", when there are no compilation or link errors from the subprogram in the file argument.
"-w"	means to suppress warnings.
"-el"	means to produce an output listing consisting of the source program being compiled interspersed with error messages (if no errors occur in the program, then no output is generated).

Tests were compiled, linked, and executed (as appropriate) using a single host computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF.

3.7.3 Test Site

Testing was conducted at Marlborough, MA and was completed on January 09, 1989.

APPENDIX A

CONFORMANCE STATEMENT

DECLARATION OF CONFORMANCE

Compiler Implementor: Encore Computer Corporation
Ada Validation Facility: NIST, Software Standards Validation Group
Ada Compiler Validation Capability (ACVC) Version: 1.9

Base Configuration

Base Compiler Name: Encore Verdix Ada Development System Version: 5.5

Host Architecture ISA:	Encore Multimax 320	OS&VER #:	Umax V, Version R2.2
Target Architecture ISA:	Encore Multimax 320	OS&VER #:	Umax V, Version R2.2

Implementor's Declaration

I, the undersigned, representing Encore Computer Corporation, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler listed in this declaration. I declare that Encore Computer Corporation is the owner of record of the Ada language compiler listed above and as such, is responsible for maintaining said compiler in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for the Ada language compiler listed in this declaration shall be made only in the owner's corporate name.

Richard T. Simpson

Date: 12/7/88

Encore Computer Corporation
Richard T. Simpson
Manager, Languages and Tools

Owner's Declaration

I, the undersigned, representing Encore Computer Corporation, take full responsibility for the implementation and maintenance of the Ada compiler listed above, and agree to the public disclosure of the final Validation Summary Report. I further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. I declare that the Ada language compiler listed, and its host/target performance is in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

John Owens

Date: 12/9/88

Encore Computer Corporation
John Owens
Vice President, Development

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the Encore Verdix Ada Development System, Version 5.5, are described in the following sections which discuss topics in Appendix F of the Ada Standard. Implementation-specific portions of the package STANDARD are also included in this appendix.

```
package STANDARD is

     ...

     type INTEGER is range -2147483648..2147483647;
     type SHORT_INTEGER is range  -32768..32767;
     type TINY_INTEGER  is range  -128..127

     type FLOAT is digits 15 range
          -1.79769313486231E+308..1.79769313486231E+308;
     type SHORT_FLOAT is digits 6 range
          -3.40282346638529E+38..3.40282346638529E+38;

     type DURATION is delta 6.10351562500000E-05 range
          -131072.0..131071.99993;

     ...
end STANDARD;
```

ATTACHMENT II

APPENDIX F. IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Pre-validation Materials/Encore VADS Version 5.5

1. IMPLEMENTATION-DEPENDENT PRAGMAS

INLINE_ONLY

This pragma, when used in the same way as pragma INLINE, indicates to the compiler that the subprogram must *always* be inlined. This pragma also suppresses the generation of a callable version of the routine which saves code space.

BUILT_IN

This pragma is used in the implementation of some predefined Ada packages, but provides no user access. It is used only to implement code bodies for which no actual Ada body can be provided, for example the MACHINE_CODE package.

SHARE_CODE

This pragma takes the name of a generic instantiation or a generic unit as the first argument and one of the identifiers TRUE or FALSE as the second argument. This pragma is only allowed immediately at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit.

When the first argument is a generic unit the pragma applies to all instantiations of that generic. When the first argument is the name of a generic instantiation the pragma applies only to the specified instantiation, or overloaded instantiations.

If the second argument is TRUE the compiler will try to share code generated for a generic instantiation with code generated for other instantiations of the same generic. When the second argument is FALSE each instantiation will get a unique copy of the generated code. The extent to which code is shared between instantiations depends on this pragma and the kind of generic formal parameters declared for the generic unit.

The name pragma SHARE_BODY is also recognized by the implementation and has the same effect as SHARE_CODE. It is included for compatibility with earlier versions of Encore VADS Ada.

NO_IMAGE

This pragma suppresses the generation of the image array used for the IMAGE attribute of enumeration types. This eliminates the overhead required to store the array in the executable image.

EXTERNAL_NAME

This pragma takes the name of a subprogram or variable defined in Ada and allows the user to specify a different external name that may be used to reference the entity from other languages. The pragma is allowed at the place of a declarative item in a package specification and must apply to an object declared earlier in the same package specification.

INTERFACE_OBJECT

This pragma takes the name of a variable defined in another language and allows it to be referenced directly in Ada. The pragma will replace all occurrences of the variable name with an external reference to the second, link_argument. The pragma is allowed at the place of a declarative item in a package specification and must apply to an object declared earlier in the same package specification. The object must be declared as a scalar or an access type. The object *cannot* be any of the following:

> a loop variable,
> a constant,
> an initialized variable,
> an array, or
> a record.

IMPLICIT_CODE

This pragma takes one of the identifiers ON or OFF as the single argument, and is only allowed within a machine code procedure. It specifies that implicit code generated by the compiler be allowed or disallowed. A warning is issued if OFF is used and any implicit code needs to be generated. The default is ON.

2. PREDEFINED PRAGMAS

CONTROLLED

This pragma is recognized by the implementation but has no effect.

ELABORATE

This pragma is implemented as described in Appendix B of the Ada RM.

INLINE

This pragma is implemented as described in Appendix B of the Ada RM.

INTERFACE

This pragma supports calls to C and FORTRAN functions. The Ada subprograms can be either functions or procedures. The types of parameters and the result type for functions must be scalar, access or the predefined type ADDRESS in SYSTEM. An optional third argument overrides the default link name. All parameters must have mode IN. Record and array objects can be passed by reference using the ADDRESS attribute.

LIST

This pragma is implemented as described in Appendix B of the Ada RM.

MEMORY_SIZE

This pragma is recognized by the implementation but has no effect. The implementation does not allow SYSTEM to be modified by means of pragmas (the SYSTEM package must be recompiled).

OPTIMIZE

This pragma is recognized by the implementation but has no effect.

PACK

This pragma will cause the compiler to minimize gaps between components in the representation of composite types. For arrays, components will only be packed to bit sizes corresponding to powers of 2, if the field is smaller than STORAGE_UNIT bits. Objects larger than STORAGE_UNIT are packed to the nearest STORAGE_UNIT. level.

PAGE

This pragma is implemented as described in Appendix B of the Ada RM.

PRIORITY

This pragma is implemented as described in Appendix B of the Ada RM.

SHARED

This pragma is recognized by the implementation but has no effect.

STORAGE_UNIT

This pragma is recognized by the implementation but has no effect. The implementation

does not allow SYSTEM to be modified by means of pragmas (the SYSTEM package must be recompiled).

SUPPRESS

This pragma is implemented as described in Appendix B of the Ada RM.

SYSTEM_NAME

This pragma is recognized by the implementation but has no effect. The implementation does not allow SYSTEM to be modified by means of pragmas (the SYSTEM package must be recompiled).

3. IMPLEMENTATION-DEPENDENT ATTRIBUTES

The attribute REF has two forms: X'REF and SYSTEM.ADDRESS(N):

In X'REF, X must be a constant, variable, procedure, function, or label. The attribute returns a value of the type MACHINE_CODE.OPERAND and may only be used to designate an operand within a code statement.

In SYSTEM.ADDRESS(N), SYSTEM.ADDRESS must be of the type SYSTEM.ADDRESS. N must be an expression of type UNIVERSAL_INTEGER. The attribute returns a value of type SYSTEM.ADDRESS, which represents the address designated by N (this is similar to the effect of an unchecked conversion from integer to address except N must be static).

4. SPECIFICATION OF PACKAGE SYSTEM

```
package SYSTEM
is

     type NAME is ( UMAX_V );

     SYSTEM_NAME      : constant NAME := UMAX_V;

     STORAGE_UNIT     : constant :=        8;
     MEMORY_SIZE      : constant := 16777216;

        -- System-Dependent Named Numbers

     MIN_INT          : constant := -2147483648;
     MAX_INT          : constant :=  2147483647;
     MAX_DIGITS       : constant :=          15;
     MAX_MANTISSA     : constant :=          31;
```

```
FINE_DELTA       : constant := 6.10351562500000E-05;
TICK             : constant := 1.00000000000000E-02;

-- Other System-dependent Declarations --

subtype PRIORITY is INTEGER range        0 .. 99;

MAX_REC_SIZE : integer :=      65536;

   type ADDRESS is private;

   NO_ADDR: constant ADDRESS;

   function PHYSICAL_ADDRESS(I: INTEGER) return ADDRESS;
   function ADDR_GT(A, B: ADDRESS) return BOOLEAN;
   function ADDR_LT(A, B: ADDRESS) return BOOLEAN;
   function ADDR_GE(A, B: ADDRESS) return BOOLEAN;
   function ADDR_LE(A, B: ADDRESS) return BOOLEAN;
   function ADDR_DIFF(A, B: ADDRESS) return INTEGER;
   function INCR_ADDR(A: ADDRESS; INCR: INTEGER) return ADDRESS;
   function DECR_ADDR(A: ADDRESS; DECR: INTEGER) return ADDRESS;

   function ">"(A, B: ADDRESS) return BOOLEAN renames ADDR_GT;
   function "<"(A, B: ADDRESS) return BOOLEAN renames ADDR_LT;
   function ">="(A, B: ADDRESS) return BOOLEAN renames ADDR_GE;
   function "<="(A, B: ADDRESS) return BOOLEAN renames ADDR_LE;
   function "-"(A, B: ADDRESS) return INTEGER renames ADDR_DIFF;
   function "+"(A: ADDRESS; INCR: INTEGER) return ADDRESS renames INCR_ADDR;
   function "-"(A: ADDRESS; DECR: INTEGER) return ADDRESS renames DECR_ADDR;

   pragma inline(PHYSICAL_ADDRESS);
   pragma inline(ADDR_GT);
   pragma inline(ADDR_LT);
   pragma inline(ADDR_GE);
   pragma inline(ADDR_LE);
   pragma inline(ADDR_DIFF);
   pragma inline(INCR_ADDR);
   pragma inline(DECR_ADDR);

private

   type ADDRESS is new INTEGER;
   no_addr: constant address := 0;

end SYSTEM
```

Attributes of the pre-defined type DURATION

first	-131072.00000
last	131071.99993
size	32
delta	6.10351562500000E-05
mantissa	31
small	6.10351562500000E-05
large	1.31071999938964E+05
fore	7
aft	5
safe_small	6.10351562500000E-05
safe_large	1.31071999938964E+05
machine_rounds	TRUE
machine_overflows	TRUE

Attributes of type FLOAT

first	-1.79769313486231E+308
last	1.79769313486231E+308
size	64
digits	15
mantissa	51
epsilon	8.88178419700125E-16
emax	204
small	1.94469227433160E-62
large	2.57110087081438E+61
safe_emax	1021
safe_small	2.22507385850720E-308
safe_large	2.24711641857789E+307
machine_radix	2
machine_mantissa	53
machine_emax	1024
machine_emin	-1021
machine_rounds	TRUE
machine_overflows	TRUE

Attributes of type SHORT_FLOAT

first	-3.40282346638529E+38
last	3.40282346638529E+38
size	32
digits	6
mantissa	21

epsilon	9.53674316406250E-07
emax	84
small	2.58493941422821E-26
large	1.93428038904620E+25
safe_emax	125
safe_small	1.17549435082228E-38
safe_large	4.25352755827077E+37
machine_radix	2
machine_mantissa	24
machine_emax	128
machine_emin	-125
machine_rounds	TRUE
machine_overflows	TRUE

Ranges of predefined integer types

TINY_INTEGER	-128 .. 127
SHORT_INTEGER	-32768 .. 32768
INTEGER	-2147483648 .. 2147483647

Default STORAGE_SIZE (collection size) for access types

100000

Priority range is 0 .. 99

Default STORAGE_SIZE for tasks is

10240

If tasks need larger stack sizes, the 'STORAGE_SIZE attribute may be used with the task type declaration.

Attributes and time-related numbers

Duration'small	6.10351562500000E-05
System.tick	1.00000000000000E-02

6. RESTRICTIONS ON REPRESENTATION CLAUSES

Pragma PACK

See section (2) above.

Size Specification

The size specification T'SMALL is not supported except when the representation specification is the same as the value 'SMALL for the base type.

7. RECORD REPRESENTATION CLAUSES

Component clauses must be aligned on STORAGE_UNIT boundaries.

Address Clauses

Address clauses are supported for objects and entries.

Interrupts

Interrupt entries are supported for UNIX signals. The Ada **for** clause gives the UNIX signal number.

Representation Attributes

The ADDRESS attribute is not supported for the following entities:

Packages
Tasks
Entries

8. MACHINE CODE INSERTIONS

Machine code insertions are supported.

The general definition of the package MACHINE_CODE provides an assembly language interface for the target machine. It provides the necessary record type(s) needed in the code statement, an enumeration type of all the opcode mnemonics, a set of register definitions, and a set of addressing mode functions.

The general syntax of a machine code statement is as follows:

CODE_N'(*opcode*, *operand* {, *operand*});

where N indicates the number of operands in the aggregate.

A special case arises for a variable number of operands. The operands are listed within a subaggregate. The format is as follows:

CODE_N'(*opcode*, (*operand* {, *operand*}));

For those opcodes that require no operands, named notation must be used (cf. RM 4.3(4)).

```
CODE_0'( op => opcode );
```

The *opcode* must be an enumeration literal (i.e. it cannot be an object, attribute, or a rename).

An *operand* can only be an entity defined in MACHINE_CODE or the 'REF attribute.

The arguments to any of the functions defined in MACHINE_CODE must be static expressions, string literals, or the functions defined in MACHINE_CODE. The 'REF attribute may not be used as an argument in any of these functions.

Inline expansion of machine code procedures is supported.

9. CONVENTIONS FOR IMPLEMENTATION-GENERATED NAMES

There are no implementation-generated names.

10. INTERPRETATION OF EXPRESSIONS IN ADDRESS CLAUSES

Address clauses are supported for constants and variables. Interrupt entries are specified with the number of the UNIX signal.

11. RESTRICTIONS ON UNCHECKED CONVERSIONS

None.

12. RESTRICTIONS ON UNCHECKED DEALLOCATIONS

None.

13. IMPLEMENTATION CHARACTERISTICS OF I/O PACKAGES

Instantiations of DIRECT_IO use the value MAX_REC_SIZE as the record size (expressed in STORAGE_UNITS) when the size of ELEMENT_TYPE exceeds that value. For example, for unconstrained arrays such as string where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead. MAX_RECORD_SIZE is defined in SYSTEM and can be changed by a program before instantiating DIRECT_IO to provide an upper limit on the record size. In any case, the maximum size supported is 64 * 1024 bytes. DIRECT_IO will raise USE_ERROR if MAX_REC_SIZE exceeds this absolute limit.

Instantiations of SEQUENTIAL_IO use the value MAX_REC_SIZE as the record size

(expressed in STORAGE_UNITS) when the size of ELEMENT_TYPE exceeds that value. For example, for unconstrained arrays such as string where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead. MAX_RECORD_SIZE is defined in SYSTEM and can be changed by a program before instantiating INTEGER_IO to provide an upper limit on the record size. SEQUENTIAL_IO imposes no limit on MAX_REC_SIZE.

14. IMPLEMENTATION LIMITS

The following limits are actually enforced by the implementation. It is not intended to imply that resources up to or even near these limits are available to every program.

Line Length

The implementation supports a maximum line length of 499 characters not including the end of line character.

Record and Array Sizes

The maximum size of a statically sized array type is 4,000,000 x STORAGE_UNITS. The maximum size of a statically sized record type is 4,000,000 x STORAGE_UNITS. A record type or array type declaration that exceeds these limits will generate a warning message.

Default Stack Size for Tasks

In the absence of an explicit STORAGE_SIZE length specification every task except the main program is allocated a fixed size stack of 10,240 STORAGE_UNITS. This is the value returned by T'STORAGE_SIZE for a task type T.

Default Collection Size

In the absence of an explicit STORAGE_SIZE length attribute the default collection size for an access type is 100,000 STORAGE_UNITS. This is the value returned by T'STORAGE_SIZE for an access type T.

Limit on Declared Objects

There is an absolute limit of 6,000,000 x STORAGE_UNITS for objects declared statically within a compilation unit. If this value is exceeded the compiler will terminate the compilation of the unit with a FATAL error message.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	1..498 => "A", 499 => "1"
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	1..498 => "A", 499 => "2"
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	1..249 => "A", 250 => "3", 251..499 => "A"
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	1..249 => "A", 250 => "4", 251..499 => "A"
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	1..496 => '0', 497..499 => '298'
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	1..493 => '0', 494..499 => '69.0E1'

$BIG_STRING1
A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.

1..199 => "A"

$BIG_STRING2
A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.

1..299 => "A", 300 => "1"

$BLANKS
A sequence of blanks twenty characters less than the size of the maximum line length.

1..479 => " "

$COUNT_LAST
A universal integer literal whose value is TEXT_IO.COUNT'LAST.

2_147_483_647

$FIELD_LAST
A universal integer literal whose value is TEXT_IO.FIELD'LAST.

2_147_483_647

$FILE_NAME_WITH_BAD_CHARS
An external file name that either contains invalid characters or is too long.

1..256 => "abcdefghijklmnop
abcdefghijklmnopabcdefghijklmnop
abcdefghijklmnopabcdefghijklmnop
abcdefghijklmnopabcdefghijklmnop
abcdefghijklmnopabcdefghijklmnop
abcdefghijklmnopabcdefghijklmnop
abcdefghijklmnopabcdefghijklmnop
abcdefghijklmnopabcdefghijklmnop
abcdefghijklmnop", 257 => "X",
258..513 => "abcdefghijklmnop
abcdefghijklmnopabcdefghijklmnop
abcdefghijklmnopabcdefghijklmnop
abcdefghijklmnopabcdefghijklmnop
abcdefghijklmnopabcdefghijklmnop
abcdefghijklmnopabcdefghijklmnop
abcdefghijklmnopabcdefghijklmnop
abcdefghijklmnopabcdefghijklmnop
abcdefghijklmnopabcdefghijklmn"

$FILE_NAME_WITH_WILD_CARD_CHAR
An external file name that either contains a wild card character or is too long.

1..256 => "0123456789ABCDEF
0123456789ABCDEF0123456789ABCDEF
0123456789ABCDEF0123456789ABCDEF
0123456789ABCDEF0123456789ABCDEF
0123456789ABCDEF0123456789ABCDEF
0123456789ABCDEF0123456789ABCDEF

$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	499
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	1..2 => '2:', 3..496 => '0', 497..499 => '11:'
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	1..3 => '16:', 4..495 => '0', 496..499 => 'F.E:'
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	1 => '"', 2..498 => 'A', 499 => '"'
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_ INT.	-2147483648
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	TINY_INTEGER
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFD#

0123456789ABCDEF0123456789ABCDEF
0123456789ABCDEF0123456789ABCDEF
0123456789ABCDEF", 257 => "X",
258..513 => "0123456789ABCDEF
0123456789ABCDEF0123456789ABCDEF
0123456789ABCDEF0123456789ABCDEF
0123456789ABCDEF0123456789ABCDEF
0123456789ABCDEF0123456789ABCDEF
0123456789ABCDEF0123456789ABCDEF
0123456789ABCDEF0123456789ABCDEF
0123456789ABCDEF0123456789ABCDEF
0123456789ABCDEF0123456789ABCD"

$GREATER_THAN_DURATION — 100_000.0
A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.

$GREATER_THAN_DURATION_BASE_LAST — 10_000_000.0
A universal real literal that is greater than DURATION'BASE'LAST.

$ILLEGAL_EXTERNAL_FILE_NAME1 — 1..256 +> "a", 257 => "1", 258..513 => "a"
An external file name which contains invalid characters.

$ILLEGAL_EXTERNAL_FILE_NAME2 — 1..256 => "b", 257 => "2", 258..513 => "a"
An external file name which is too long.

$INTEGER_FIRST — -2147483648
A universal integer literal whose value is INTEGER'FIRST.

$INTEGER_LAST — 2147483647
A universal integer literal whose value is INTEGER'LAST.

$1NTEGER_LAST_PLUS_1 — 21474836478
A universal integer literal whose value is INTEGER'LAST + 1.

$LESS_THAN_DURATION — -100_000.0
A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.

$LESS_THAN_DURATION_BASE_FIRST — -10_000_000.0
A universal real literal that is less than DURATION'BASE'FIRST.

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 28 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

B28003A: A basic declaration (line 36) wrongly follows a later declaration.

E28005C: This test requires that 'PRAGMA LIST (ON);' not appear in a listing that has been suspended by a previous "pragma LIST (OFF);"; the Ada Standard is not clear on this point, and the matter will be reviewed by the ARG.

C34004A: The expression in line 168 wrongly yields a value outside of the range of the target type T, raising CONSTRAINT_ERROR.

C35502P: Equality operators in lines 62 & 69 should be inequality operators.

A35902C: Line 17's assignment of the nomimal upper bound of a fixed-point type to an object of that type raises CONSTRAINT_ERROR, for that value lies outside of the actual range of the type.

C35904A: The elaboration of the fixed-point subtype on line 28 wrongly raises CONSTRAINT_ERROR, because its upper bound exceeds that of the type.

C35904B: The subtype declaration that is expected to raise CONSTRAINT_ERROR when its compatibility is checked against that of various types passed as actual generic parameters, may in fact raise NUMERIC_ERROR or CONSTRAINT_ERROR for reasons not anticipated by the test.

C35A03E, & R: These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard doesn't support this assumption.

C37213H: The subtype declaration of SCONS in line 100 is wrongly expected to raise an exception when elaborated.

C37213J: The aggregate in line 451 wrongly raises CONSTRAINT_ERROR.

C37215C, E, G, H: Various discriminant constraints are wrongly expected to be incompatible with type CONS.

C38102C: The fixed-point conversion on line 23 wrongly raises CONSTRAINT_ERROR.

C41402A: 'STORAGE_SIZE is wrongly applied to an object of an access type.

C45332A: The test expects that either an expression in line 52 will raise an exception or else MACHINE_OVERFLOWS is FALSE. However, an implementation may evaluate the expression correctly using a type with a wider range than the base type of the operands, and MACHINE_OVERFLOWS may still be TRUE.

C45614C: REPORT.IDENT_INT has an argument of the wrong type (LONG_INTEGER).

E66001D: AI-330 states this test is to be changed from an "E" test to a "B" test during the next version of the ACVC. AI-330 was approved in July 1986, 6 months before the initial version of ACVC Version 1.9 was released and a nearly a full year before the final version of ACVC Version 1.9 was released. This test is withdrawn pending further comment from AJPO regarding issue of the test being a B Test rather than an E Test.

A74106C, C85018B, C87B04B, CC1311B: A bound specified in a fixed-point subtype declaration lies outside of that calculated for the base type, raising CONSTRAINT_ERROR. Errors of this sort occur re lines 37 & 59, 142 & 143, 16 & 48, and 252 & 253 of the four tests, respectively (and possibly elsewhere).

BC3105A: Lines 159..168 are wrongly expected to be illegal; they are legal.

AD1A01A: The declaration of subtype INT3 raises CONSTRAINT_ERROR for implementations that select INT'SIZE to be 16 or greater.

CE2401H: The record aggregates in lines 105 & 117 contain the wrong values.

CE3208A: This test expects that an attempt to open the default output file (after it was closed) with mode IN_FILE raises NAME_ERROR or USE_ERROR; by Commentary AI-00048, MODE_ERROR should be raised.